Intrinsic currying for C++ template metaprograms

Keir, Paul; Gozillon, Andrew; Haeri (Hossein), Seyed H.

Published in:
Trends in Functional Programming

DOI:
10.1007/978-3-030-18506-0_3

Published: 24/04/2019

Document Version
Peer reviewed version

This is a post-peer-review, pre-copyedit version of an article published in Trends in Functional Programming: 19th International Symposium, TFP 2018 Gothenburg, Sweden, June 11–13, 2018 Revised , Sweden, June 11-13, Selected Papers.

The final authenticated version is available online at: http://dx.doi.org/10.1007/978-3-030-18506-0_3

Intrinsic Currying for C++ Template Metaprograms

Paul Keir[1][0000−0002−4781−9377], Andrew Gozillon[1][0000−0001−7558−7166], and Seyed H. HAERI (Hossein)[2][0000−0002−7969−8573]

[1] University of the West of Scotland, Paisley, UK
{paul.keir,andrew.gozillon}@uws.ac.uk
[2] Université catholique de Louvain, Louvain-la-Neuve, Belgium
hossein.haeri@uclouvain.be

Abstract. C++ template metaprogramming is a form of strict functional programming, with a notable absence of intrinsic support for elementary higher-order operations. We describe a variadic template metaprogramming library which offers a model of implicitly curried, left-associative metafunction application through juxtaposition; inspired by languages such as Haskell, OCaml and F♯. New and existing traits and metafunctions, constructed according to conventional idioms, seemlessly take advantage of the framework's features. Furthermore, a distinctive versatility is exposed, allowing a user to define higher-order metafunction classes using an equational definition syntax; without recourse to elaborate nested metafunctions. The primary type expression evaluator of the library is derived from a single application of an elementary folding combinator for type lists. The definition of the fold's binary operator argument is therefore a focal point; and constructed mindful that substitution failure of a template parameter's deduced type produces no compilation error. Two distinctive features of C++ metafunctions require particular consideration: zero argument metafunctions; and variadic metafunctions. We conclude by demonstrating characteristics of the library's main evaluation metafunction in conjunction with the universal property of an updated *right-fold* combinator, to compose a range of metafunctions including *map*, *reverse*, *left-fold*, and the *Ackermann function*.

Keywords: Types · Templates · Metaprogramming · Currying

1 Introduction

C++ template metaprogramming is a form of strict functional programming, with a notable absence of intrinsic support for elementary higher-order operations. Having no canonical representation of metafunctions, authors of template metaprogramming libraries are left to endlessly reinvent the wheel. In this paper we argue that the development of C++ metaprogramming requires a further component: implicit currying.

Beyond basic parametric polymorphism, interesting template metaprograms make fullsome use of a Turing complete language, and rely on *explicit* recursion for full expressivity; yet the advantages of *structured* recursion schemes such as catamorphisms (folds) are widely understood [24].

Considering the high profile of C++, and template metaprogramming, it is surprising that substantial use of template metaprogramming is so rare. Despite notable exceptions [12,20,14,33,40,21], the opportunity to enforce program correctness at compile-time, through embedded domain-specific languages, is underexplored.

Template metaprogramming libraries certainly exist [4,9,27,17,10] and are utilised widely. Many basic templates offered by these libraries were added to the C++11 [2] version of the language standard; including for example a simple type wrapper for integral constants. Prior to this, libraries such as the Boost Metaprogramming Library (MPL) [4] were an obvious choice. Boost itself is a pre-eminent collection of over 150 open-source C++ libraries. Of these, over 40 make use of MPL including: Proto, for creating embedded domain-specific libraries (EDSLs); Spirit, a parser framework; Fusion, a tuple library; Phoenix, a functional programming interface; Metaparse, for parsing strings at compile time; Graph, a generic graph-traversal library; and Hana [9], a more recent metaprogramming library, which is itself now used by Boost Yap, a C++14 expression template library. So, as elementary idioms become included within successive C++ standards, new concepts blossom in fresh libraries. Nevertheless, obstacles inhibit wider adoption; including the apparent complexity of the discipline.

There are likely a number of reasons for this. Poor syntax is often cited; for example dependent name disambiguation via the `template` and `typename` keywords. So too, while metaprogramming libraries provide structured recursion operators[3] such as *map* and *fold*, standard C++ library support is regretably absent. Idiomatic functional programs will of course also utilise the higher order nature of such operators, and yet no standard C++ representation exists for metafunctions; nor even a standard approach for *returning* a metafunction.

Finally on this point, consider the effort of translating a sizable functional program into a C++ metaprogram without currying. Even the simplest tutorial on recursive combinators such as *map* will soon introduce code such as: *(map (1+) xs)*; yet notice the use of implicit currying within the *section* of the infix addition operator in *(1+)*. Such missing features are a significant inhibiting factor in the pursuit of good practice, including *reuse*, in C++ metaprogramming. The *Curtains* library has been developed to address such concerns; facilitating an embedded domain-specific language for C++ template metaprograms, with support for implicit currying. The Curtains equivalent of the Haskell expression *(const map () (1+) [0,1,2])* is shown below:

```
eval<const_q,map_q,void,eval<add_q,ic<1>>,ilist<0,1,2>>
```

[3] Often *map* is named `transform` after the standard C++ runtime function. A *fold* is more often named `fold`; with `accumulate` provided as an alias.

We present *three* implementations, each built upon a single recursive combinator: a left-fold; and while each implementation is purposefully distinct, differences are accounted for entirely by the choice of higher-order, binary combining operation used with the fold. Of the three approaches, our preferred is accomplished in just 30 lines of code, and handles fixed arity metafunctions. The second approach handles idiomatic variadic and nullary metafunctions; while the third treatment finds a middle way, by asking the user to select a single arity, for an otherwise possibly variadic metafunction. Each implementation supports implicit currying.

2 Elementary Metaprogramming

In C++ a user-defined type is referred to as a *class*, yet may be declared using either the `class` keyword, or the `struct` keyword. The difference between the two forms relates only to the default access permission of the class. For ease of exposition we will use the latter, more permissive form throughout; so avoiding verbose usage of the `public` access specifier. Following the class declaration shown below, the type expression `int_wrap::type` becomes synonymous with `int`.

```
struct int_wrap { using type = int; };
```

A class may also be parameterised, and so declared as a *class template*. The class template `add_pointer`, shown below, is parameterised with a single *type* template parameter[4] named `T`. Providing `add_pointer` with a type argument *instantiates* the template; so forming a *type*. The resulting type, say `add_pointer<int>`, may then be used wherever a type is expected; say to declare a runtime variable, or as an argument for another template.

```
template <class T> struct add_pointer { using type = T*; };
```

A common metaprogramming idiom can then be explained. A class template with a single member type definition, conventionally names the member: `type`. Compile-time class template parameters can then be understood as isomorphic to common run-time function parameters; with the relevant member type definition analogous to the return value. A class template so equipped is often referred to as a *metafunction*.

From this perspective, the `add_pointer` *type trait* class template from the standard C++ type support library is a *unary* metafunction. Given an `int` argument, the metafunction returns a *first order* type; an `int*`, within the `type` member of the instantiated `add_pointer` template. The application of such a metafunction will involve the familiar angle-bracket syntax: `add_pointer<int>`; with the result obtained via `typename add_pointer<int>::type`[5].

[4] This is a second, distinct usage of the `class` keyword. The `struct` keyword is not permitted in this context; though `typename` is.

[5] The `typename` disambiguator informs the compiler that a dependent name following the `::` operator, refers to a *type* [38, p. 228].

Template metaprograms are untyped; though various mechanisms exist to allow ad-hoc treatment of particular types, or type patterns, via class template *specialisation*. For example, the specialisation on the second line of the following possible implementation of the standard C++ library type trait, `remove_const`, handles types which are `const` qualified; so matching the type pattern: `const T`.

```
template <class T> struct remove_const          { using type = T; };
template <class T> struct remove_const<const T> { using type = T; };
```

A crucial component of elementary first-order template metaprogramming is recursion. One or more class template specialisations can represent the *base cases*. Meanwhile each *recursive step* includes an instantiation of the class template being defined. We demonstrate recursion using integers at the type level; with the assistance of the `ic` *alias template* defined below[6]. Akin to a C++ `typedef`, or a Haskell type synonym, an alias template defines a new name for an existing template: `std::integral_constant` in this case. On *this* occasion, `ic` specifies a *non-type* parameter; and the `auto` specifier ensures the argument's type is inferred. Consequently, the *type* `ic<42>`, for example, can concisely represent the word-sized compile-time integer constant: 42.

```
template <auto I> using ic = std::integral_constant<decltype(I),I>;
```

Using our integer representation, the code below defines a recursive template metafunction, `fact`, which calculates the factorial of its argument[7]. For example `fact<ic<3>>::type` ≡ `ic<6>`.

```
template <class T> struct fact;
template <       > struct fact<ic<0>> : ic<1> {};
template <auto  N> struct fact<ic<N>> :
                                ic<N*typename fact<ic<N-1>>::type{}> {};
```

The standard C++ type support library defines an alias template for each type trait, providing a convenient syntax to access the `type` member of the associated class template. For example, `add_pointer_t<int>` evaluates to `int*` using the `add_pointer_t` alias template shown below:

```
template <class T> using add_pointer_t = typename add_pointer<T>::type;
```

Specialisation of alias templates is not possible; and neither is *recursion*. Consequently, alias templates have limited capability, and are typically used to provide *syntactic sugar* to existing class template definitions. Alias templates can nevertheless themselves be interpreted as metafunctions.

Lastly, *variadic templates* facilitate a variable quantity of template arguments. For example, the variadic alias template `ct_tail` below accepts one or

[6] `decltype` is a keyword used to query the type of an expression.

[7] The type expression `typename fact<ic<N-1>>::type` will evaluate to a type; an instantiation of `ic`, and hence also of `std::integral_constant`. The `{}` braces which follow this expression will aggregate-initialise a `constexpr std::integral_constant` object before using its conversion operator member to provide an `int` value as the multiplier, with `N` the multiplicand.

more arguments: e.g. `ct_tail<char*,int,long>` ≡ `long`. The standard library's `std::common_type_t` obtains a common type from its parameter pack argument[8].

```
template <class T, class... Ts> using ct_tail = std::common_type_t<Ts...>;
```

2.1 Higher Order Metaprogramming

A basic challenge for higher-order template metaprogramming is how a metafunction should be *returned*. Metafunctions can certainly be passed as template *arguments*. Considering the class template definition `ho`, below, `ho<add_pointer>` and `ho<add_pointer_t>` are both valid template instantiations.

```
template <template <class> class> struct ho {};
```

Given that a metafunction can be defined *either* using an alias template; or a class template, it is reassuring that either approach can also represent the return types: a metafunction can either "return" a nested class template; or an alias template. The code below defines a unary metafunction, `ct`, which returns another metafunction; as the member alias template `m_invoke`. Applying the `ct` metafunction to a type argument, will thus return a metafunction which itself determines the common type among the arguments that it is provided *and* the single argument already provided to `ct`.

```
template <class T>
struct ct {
  template <class... Ts>
  using m_invoke = std::common_type_t<T,Ts...>;
};
```

Applying the metafunction returned by `ct` to a `long` argument, might involve syntax such as the following: `typename ct<long>::template m_invoke<int>`[9]; a type expression which evaluates to `long`. The verbosity of such nested metafunction invocations can be reduced through a helpful combinator:

```
template <class F, class... Ts>
using invoke = typename F::template m_invoke<Ts...>;
```

The `invoke` combinator offers improved syntax when applying a nested metafunction member named `m_invoke`; with the previous example represented as `invoke<ct<long>,int>`. Applying a returned metafunction can thus both be transcribed more concisely; while clearly articulating the separation of a nested metafunction from its arguments. Without further treatment, however, *non-nested* metafunctions, such as type traits, are incompatible. The first argument of the `invoke` combinator, expects a *type*; not a *template*. A combinator to envelop the functionality of an arbitrary type trait template, within the `m_invoke` alias template member of a proxy class, would be useful. Consider `quote` below:

[8] Regarding a template parameter pack: note that its *declaration* is *preceded* by an ellipsis; while its *expansion* is *followed* by one [38, p. 188].

[9] The `template` disambiguator informs the compiler that a dependent name following a `::`, `->`, or `.` operator, refers to a *template* [38, p. 230].

```
template <template <class...> class M>
struct quote {
  template <class... Ts>
  using m_invoke = M<Ts...>;
};
```

Instantiating `quote` with a suitable class template argument, `M`, will produce a type with an `m_invoke` member metafunction, and equivalent functionality to the `M` argument. Given a type trait *template* such as `std::common_type_t`, for example, the `quote<std::common_type_t>` *type* is a suitable first argument for the `invoke` combinator; and the following equivalency between types holds:

```
invoke<quote<common_type_t>,int,long> ≡ long
```

Alas, instantiating `quote` with a *non-variadic* template argument, as in `quote<std::add_pointer_t>`, will produce a compilation error; relating to the provision of a pack argument to a non-pack template parameter list. The interface and functionality of `quote` can nevertheless be constructed in unambiguous C++ using an alternative approach.

Expanding a parameter pack into a fixed length template parameter list is problematic *only* within the context of an *alias* template[10]; as with the `m_invoke` member of `quote`. By ensuring the parameter pack is instead expanded within a *class* template, this problematic and byzantine corner case can be evaded. An additional class template, `iv1430`[11], is therefore introduced:

```
template <class, template <class...> class, class...>
struct iv1430 {};

template <template <class...> class M, class... Ts>
struct iv1430<void_t<M<Ts...>>,M,Ts...> { using type = M<Ts...>; };
```

A final version of `quote`[12] can then be defined, using `iv1430`, as shown below. With this version, no compilation errors are encountered when `quote` is provided with non-variadic template arguments, as pack expansion now occurs within a class template rather than an alias template. For example, non-variadic traits such as `std::add_pointer_t` or `std::is_object`; as well as variadic ones such as

[10] Discussion regarding the virtue of this restriction remains an active topic within the ISO C++ Standards Committee; identified as core issue 1430 [25].

[11] The `iv1430` class template makes use of a powerful C++ feature wherein template substitution failure is not an error (SFINAE). Furthermore, `iv1430` is defined as "SFINAE-friendly". For example, as `int` and `int*` have no *common type*, neither `common_type<int,int*>`, nor the equivalent `iv1430<void,common_type_t,int,int*>`, has a `type` member. The failed instantiation of the `iv1430` specialisation with `iv1430<void,common_type_t,int,int*>` resolves to the primary template; leaving subsequent access to the absent `type` member *only* a substitution error. The alternative is a hard error. C++17's `std::void_t` is an alias template helpful in such contexts; provided with zero or more *valid* type template arguments, the aggregate instantiates to `void` [38, p. 420].

[12] A variant, `quote_c`, accepting class template arguments, is defined in Appendix A.1.

`std::conjunction` or `std::is_constructible` can each be converted, using `quote`, into a form suitable for use with the `invoke` combinator. Such a type, suitably equipped with an `m_invoke` alias template member, either by the use of the `quote` combinator, or by elementary design, is known as a *metafunction class*.

```
template <template <class...> class M>
struct quote {
  template <class... Ts>
  using m_invoke = typename iv1430<void,M,Ts...>::type;
};
```

The following two equivalences demonstrate the use of `invoke` along with the non-variadic metafunctions, `std::add_pointer` and `std::add_pointer_t`:

```
invoke<quote<std::add_pointer_t>,int> ≡ int*
invoke<quote<std::add_pointer  >,int> ≡ add_pointer<int>
```

The intermediate reduction steps for the first of these is shown below:

```
invoke<quote<std::add_pointer_t>,int>
   ≡ {invoke alias template}
quote<std::add_pointer_t>::m_invoke<int>>
   ≡ {m_invoke alias template member of quote template}
iv1430<void,std::add_pointer_t,int>::type
   ≡ {type member of iv1430 template specialisation}
std::add_pointer_t<int>
   ≡ {standard C++ alias template add_pointer_t}
std::add_pointer<int>::type
   ≡ {standard C++ template add_pointer}
int*
```

2.2 The Identity Metafunction

A metafunction which returns its argument, is a central component throughout the Curtains implementation. The `id` metafunction, shown below, utilised as a *mix-in* class template, can specify a base class, facilitating the common requirement within metaprogramming for a class to include a member type definition named `type`; introduced orthogonally here via inheritance.

```
template <class T>  struct id { using type = T; };
```

A possible implementation of `std::add_volatile`, constructed using this idiom, is shown below. Seen as a metafunction, such syntax can be interpreted as highlighting the *type* which will be provided as the return "value"; located to the right of the colon, within the angle brackets of the `id` template.

```
template <class T> struct add_volatile : id<volatile T> {};
```

With routine application of the `invoke` combinator, providing a type trait as an argument to the `quote` combinator can become as common as accessing a `type` member. We adopt the "`_q`" suffix here in deference to the "`_t`" suffix convention

of the standard C++ type support library's alias templates. The relevant pair for the `add_volatile` type trait are shown in the code below:

```
template <class T>
using add_volatile_t = typename add_volatile<T>::type;
using add_volatile_q = quote<add_volatile_t>;
```

3 Curried Template Evaluation

This section presents our implicitly currying evaluation mechanism: the metafunction `eval`.

Function application in Haskell [22] is written *e1 e2*; where *e2* is an arbitrary Haskell expression, and *e1* is a Haskell expression which reduces to a value with a function type. Application associates to the left, and so the parentheses may be omitted in *(f x) y*. Hence function application is implicitly curried within Haskell. We aim to create a comparable evaluation environment within the context of C++ metaprogramming. Given a C++ variadic template evaluator, *eval*, we would like metafunction application to be written *eval<e1,e2[,...]>*, where the ellipsis represents an optional trailing list of type arguments. Metafunction application should also associate to the left, and hence the omission of the inner template instantiation of `eval` within `eval<eval<F,X>,Y>` would be permitted; and denoted as `eval<F,X,Y>`.

A basic expectation is that a quoted metafunction, provided to `eval`, together with a full set of valid template arguments, should produce the same result as with the traditional metafunction alone. For an arbitrary metafunction *M*, and parameter pack *Ts*, where *M<Ts...>* is well formed; *eval<quote<M>,Ts...>* is also well formed. Furthermore, the following equality holds:

$$M\texttt{<}Ts\texttt{...>} \equiv eval\texttt{<}quote\texttt{<}M\texttt{>,}Ts\texttt{...>}$$

For example, given `add_pointer_q`, defined as `quote<std::add_pointer_t>`, we find that `eval<add_pointer_q,int>` ≡ `std::add_pointer_t<int>` ≡ `int*`. Class templates are also suitable metafunctions; and so `eval<quote<std::is_pod>,int>` ≡ `std::is_pod<int>`. Note that the equality assumes a valid left-hand side; for curried applications of *quote<M>*, only the right-hand side will be valid.

3.1 Components of Implicit Currying

Given such conditions, it follows that a method for managing the partial evaluation of a metafunction is required. The `curry` class template below can help here: given a metafunction class `F` as its *first* argument, and n further type arguments, `curry` will instantiate to a new metafunction class; equivalent to `F` partially applied to those n arguments. Additional arguments can be provided using `curry` again; or `invoke` (page 5) may be used to instantiate the full metafunction application, and optionally also supply any remaining arguments.

```
template <class F, class... Ts>
struct curry {
  template <class... Us>
  using m_invoke = invoke<F,Ts...,Us...>;
};
```

The equivalences below demonstrate uses of `curry` with a metafunction class `common_type_q`; constructed via `quote` from the type trait: `std::common_type_t`. Note `invoke`'s accommodation of nested instantiations of `curry`, here; and referenced later in Section 4.2.

```
invoke<curry<common_type_q,char>,int>              ≡ int
invoke<curry<common_type_q,char,int>,long>         ≡ long
invoke<curry<curry<common_type_q,char>,int>,long> ≡ long
```

The C++ compiler will issue a helpful error if the first argument to `curry` is not a metafunction class; due to the instantiation of `invoke` within `curry`'s `m_invoke` member. Error checking is nevertheless incidental; and providing "too many" arguments for `curry`'s template parameter pack *is* accepted, at least until its application through `invoke`.

Note that the absence of currying within the simple `invoke`, will manifest itself in a compilation error, and *not only* with metafunctions applied to "too few" arguments; as in say `invoke<quote<is_same>,int>`. In Haskell *(id id 42)* ≡ *((id id) 42)*; and either expression thus evaluates to 42. Consider the definition of `id_q` below, a simple preparation of the earlier `id` template class:

```
using id_q = quote_c<id>;
```

A comparable C++ template expression, `invoke<id_q,id_q,ic<42>>`, will also result in an error; as `invoke` attempts to apply its first argument, to all those that remain; "too many" arguments. So too the issue may not be as conspicuous; while the Haskell expression *(foldr id 42 [id])* presents *foldr* with its full quota of three arguments, reduction will nevertheless require evaluation of the familiar *(id id 42)*.

One may briefly consider the remedy of prescribing thorough use of the `curry` class template. However, while explicit invocation of an occasional application of `curry` may be tolerable, *systematic* integration within a larger system would be tedious; and error prone as a consequence. This is the explicit currying seen in other systems.

3.2 Folding with Types

The creation of a C++ metafunction expression evaluator, with *implicit* currying, can be achieved by defining both: a generic folding combinator; and a specific combining operation. Adopting Haskell's model of function application involves the left-associative currying operator, denoted by the space between operands; i.e. their *juxtaposition*. Given the elementary binary function *const*, which returns its first argument, the Haskell-like expression *(const 1 2)* is parsed as *((const 1) 2)*, with *(const 1)* returning a function equivalent to the partial

application of *const*. Operationally, this can be processed as a *left-fold*, with currying as the binary combining operation. A definition for a similar fold over a homogeneous list in Haskell is shown below:

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Evaluating an expression such as *(const 1 2)* can then be understood intuitively as a left-fold on a heterogeneous list of three elements: the binary function *const*; the numeric literal *1*; and the numeric literal *2*.

The code below defines a C++ left-fold metafunction class[13] through two specialisations of a class template `ifoldl`; one for the base case; and one for the recursive step. The type template parameter `F`, which expects a metafunction class, is seen applied to two arguments, `Z` and `T`, at the `invoke<F,Z,T>` instantiation within the recursive `ifoldl` specialisation. Finally, the `quote_c` alias template is used to produce a metafunction class suitable for use with `invoke`: `ifoldl_q`.

```
template <class, class Z, class...>
struct ifoldl              : id<Z> {};

template <class F, class Z, class T, class... Ts>
struct ifoldl<F,Z,T,Ts...> : ifoldl<F,invoke<F,Z,T>,Ts...> {};

using ifoldl_q = quote_c<ifoldl>;
```

The left-fold above is defined recursively according to conventional metaprogramming idioms. As a simple example of its operation, the code below performs a compile-time calculation of $((0 - 1) - 2)$.

```
template <class T, class U> using sub = ic<T::value - U::value>;

invoke<ifoldl_q,quote<sub>,ic<0>,ic<1>,ic<2>> ≡ ic<-3>
```

It is noteworthy that an implementation utilising C++17's *fold expressions*, with an equivalent interface and functionality is also possible, though no more concise, by overloading an arbitrary binary operator. Such a version is provided in Appendix A.11.

While considering the definition of a combining operation for use with the folding metafunction class, `ifoldl_q`, and with which to facilitate an implicitly currying evaluator, it can be worthwhile to examine the limitations of naively providing `ifoldl_q` with a suitably *quoted* version of either `invoke`; or `curry`. Corresponding evaluators, `eval_i` and `eval_c`, are shown below:

```
template <class F, class... Ts>
using eval_i = invoke<ifoldl_q,quote<invoke>,F,Ts...>;

template <class F, class... Ts>
using eval_c = invoke<ifoldl_q,quote<curry>, F,Ts...>;
```

[13] A template parameter pack `Ts` is used by `ifoldl` rather than a type list; and hence it is not strictly isomorphic to the Haskell fold above; this is however only part of the implementation, *not* of the public API.

The `eval_i` combinator will apply `invoke`, two arguments at a time, starting from the leftmost pair. Of course as `invoke` has no support for currying, this only succeeds for unary functions. For example, `eval_i<id_q,id_q,ic<42>>`, akin to the Haskell expression, *(id id 42)*, *will* reduce to `ic<42>` as expected. However, `eval_i<const_q,int,bool>`, in attempting to instantiate `invoke<const_q,int>`, instead produces a compilation error; `const_q` is defined below:

```
template <class T, class> using const_t = T;
                          using const_q = quote<const_t>;
```

The `eval_c` combinator will instantiate nested `curry` classes instead; ever deeper with recursive step. A final application of `invoke` may then be useful, to convert the nested `curry` classes, into the expected result type. For example, `eval_c<const_q,int,bool>` will reduce to `curry<curry<const_q,int>,bool>`, which *is* a valid metafunction class; providing it to `invoke`, with no further arguments, will produce the anticipated result: `int`. The `eval_c` combinator, combined with the final application of `invoke`, thus behaves exactly as `invoke` alone. Consequently, it encounters the same restrictions regarding currying; for example `eval_c<id_q,id_q,ic<42>>` fails to compile as `id_q` is provided with two arguments; `id_q` and `ic<42>`. Another approach is required.

A *suitable* binary combining operation for use with the left-fold of `ifoldl_q` makes conditional use of both `curry` *and* `invoke`. Algorithm 1 illustrates in pseudocode the operation of this metafunction; with the C++ definition provided below. Intuitively, `curry_invoke_q` will use `invoke` to apply its first argument to its second, when possible; otherwise it returns the application in curried form.

```
template <class F, class T, class = void_t<>>
struct curry_invoke                          : id<curry<F,T>>  {};

template <class F, class T>
struct curry_invoke<F,T,void_t<invoke<F,T>>> : id<invoke<F,T>> {};

using curry_invoke_q = quote_c<curry_invoke>;
```

With `curry_invoke_q` as the combining operation of `ifoldl_q`, and the elementary `id_q` as a starting value, an implicitly currying evaluation metafunction, `eval`, can be defined; as shown below.

```
template <class... Fs>
using eval = invoke<ifoldl_q,curry_invoke_q,id_q,Fs...>;
```

As `eval` is defined by a conventional catamorphism, `eval<>` simply returns the "zero" value of the defining left-fold; the identity metafunction class: `id_q`. Likewise, for an arbitrary type *T*, *`eval<T>`* evaluates to *T*; consequently `eval<int>` and `eval<id_q>` reduce to `int` and `id_q` respectively. Demonstrations of the utility of `eval` are explored in Section 5; while the intermediate steps involved in reducing a sample expression, `eval<const_q,id_q,int,char>`, to `char` are listed in Appendix A.12.

Algorithm 1 Invocation with conditional currying

Precondition: f is a possibly curried metafunction class
Precondition: t is an arbitrary type
Postcondition: g is a curried metafunction class

```
1: function CURRY-INVOKE(f, t)
2:     if ISVALIDEXPRESSION(f(t)) then
3:         g ← f(t)
4:     else
5:         g ← CURRY(f, t)
6:     end if
7:     return g
8: end function
```

4 Variadic and Nullary Metafunctions

Section 3's `eval` metafunction accommodates a domain-specific language of expressions involving curried evaluation of *fixed arity* metafunctions. C++ templates also support idiomatic *nullary* and *variadic* metafunctions. Variadic templates were introduced in C++11. While the argument count and values of the instantiated template may be unknown to a template's author, such aspects are of course resolved at compile-time. Meanwhile, nullary metafunctions arise when either the template parameters of a class are specified with default values; or a variadic class template has a template parameter pack, optionally with preceding defaulted template parameters. For an arbitrary nullary metafunction, `N`, angle brackets remain necessary during instantiation; as in `N<>`. We propose that a modified evaluation combinator, `eval_v`, support the following syntax for such eventualities: `eval_v<N>`. Our implementation will continue to operate as a left-fold, but more intricacy and heuristics is required for the combining operation.

4.1 An Antidetection Idiom

Now is an opportune moment to introduce a novel metafunction: `invalid`; a combinator aligned with the SFINAE *detection idiom* [7], which provides a useful form of inverse to the idiomatic application of C++17's `std::void_t`.

An elementary enquiry, facilitated easily using `std::void_t`, is whether or not a class has a member named `type`. Instantiating the `xt` class template shown below using a type which *does* have such a member, will create a type which *does not*; and vice versa: instantiating `xt` with an argument which *does not* have a `type` member, will produce a type which does.

```
template <class, class = void_t<>>
struct xt : id<void>                  {};

template <class T>
struct xt<T,void_t<typename T::type>> {};
```

For an arbitrary metafunction class `F`, and type arguments `Ts...`, where `void_t<invoke<F,Ts...>>` instantiates to `void`, we can interpret the argument to `void_t` as being *valid*. Using the `invalid` combinator, shown below, with `F` and `Ts` arguments as before, will see `void_t<invoke<invalid<F>,Ts...>>` *fail* to instantiate; where `invoke<F,Ts...>` is valid, `invoke<invalid<F>,Ts...>` is *invalid*.

```
template <class F>
struct invalid {
  template <class... Us>
  using m_invoke =
    typename xt<iv1430<void,F::template m_invoke,Us...>>::type;
};
```

Considering the potential for multiple arguments to `std::void_t`, the idea emerges to combine requirements for valid instantiations of `invoke` with those which are invalid. For example, for arbitrary types `T` and `U`, `void_t<invoke<F,T>,invoke<invalid<F>,U>>` could help specify that a class template specialisation should be selected when `invoke<F,T>`, but *not* `invoke<F,U>`, is valid.

4.2 The Combining Operation

Algorithm 2 Conditional invocation of a function and argument

Precondition: f is a possibly curried metafunction class
Precondition: t is an arbitrary type
Postcondition: g is a curried metafunction class

```
1: function Curry-invoke-peek(f, t)
2:     if IsValidExpression(f()) ∧ ¬IsValidExpression(f(t)) then
3:         f′ ← f()
4:         g ← Curry-invoke-peek(f′, t)
5:     else
6:         g ← Curry(f, t)
7:     end if
8:     return g
9: end function
```

A new binary combining operation for use with Section 3.2's left-fold again makes conditional use of `curry` *and* `invoke`; with additional SFINAE guidance from the `invalid` class template. Algorithm 2 illustrates in pseudocode the operation of this recursive metafunction. On line 2, the first argument of *Curry-invoke-peek*, a function `f`, perhaps already with curried arguments, has its potential for application determined, both: (1) with no further arguments; and (2) with `t` as a single argument. Having the first condition true, with the second false, allows for `f` to be evaluated, with the resulting function `f'` passed alongside `t`, via a recursive call to *Curry-invoke-peek* on line 4. Alternatively, the *Curry-invoke-peek*

function will conclude on line 6 simply by returning *f*, curried with the type *t*. The C++ definition, curry_invoke_peek_q, is shown below.

```
template <class F, class T, class = void_t<>>
struct curry_invoke_peek : id<curry<F,T>> {};

template <class F, class T>
struct curry_invoke_peek<F,T,void_t<invoke<F>,invoke<invalid<F>,T>>>
     : curry_invoke_peek<invoke<F>,T> {};

using curry_invoke_peek_q = quote<curry_invoke_peek>;
```

Algorithm 2 differs from algorithm 1 in two ways. Firstly, `invoke` may be used *either* to evaluate the application of *f* to *t*; *or*, to evaluate *f* itself. This supports variadic metafunctions, through the incremental consideration of additional arguments. Secondly, algorithm 2 is recursive; accommodating nullary metafunctions, which too may return nullary metafunctions.

Given a metafunction class *F*, and a template parameter pack *Fs*, the type expression *invoke<ifoldl_q,curry_invoke_peek_q,F,Fs...>* will produce a *curried* representation of a metafunction application. Consider the invocation of `invoke` below; comparable to the Haskell *(const id 42 'a')*.

```
invoke<ifoldl_q,curry_invoke_peek_q,const_q,id_q,int,char>
```

The expression reduces to `curry<id_q,char>`. Applying `invoke` to this, once again, with no further arguments, can produce the sought `char`. A curried metafunction application, with fewer arguments than required by the metafunction, is also a valid output of the fold; say `curry<const_q,int>`. Applying `invoke` to this, however, produces a compilation error. A metafunction combinator is thus defined, which applies `invoke` to a curried metafunction application *only* when this can be achieved without error.

The `invoke_if` metafunction combinator defined below will apply `invoke` conditionally to its `F` parameter whenever possible; otherwise, `F` is simply returned. Furthermore, should such an invocation be achieved, the attempt will be repeated; by passing the result, `invoke<F>`, recursively to `invoke_if`. By this route, the uncommon scenario of a nullary metafunction returning a possibly nullary metafunction is also handled.

```
template <class F, class = void_t<>>
struct invoke_if                      : id<F>                {};

template <class F>
struct invoke_if<F,void_t<invoke<F>>> : invoke_if<invoke<F>> {};

template <class F> using invoke_if_t = typename invoke_if<F>::type;
```

Consequently, `eval_v` can be defined as shown below. As discussed, the fold produces a curried result; hence the `invoke_if` combinator, via `invoke_if_t`, is utilised to conditionally apply `invoke` upon it.

```
template <class... Fs>
using eval_v =
  invoke_if_t<invoke<ifoldl_q,curry_invoke_peek_q,id_q,Fs...>>;
```

Evaluating metafunction applications involving nullary metafunctions can produce surprising results. Consider the `zero_constv_q` metafunction class below:

```
template <class...> struct zero_constv : id<const_q> {};

using zero_constv_q = quote_c<zero_constv>;
```

Applied to *zero* arguments, the type expression `eval<zero_constv_q>`, reduces to `const_q`. In fact for an arbitrary pack of types *`Ts`*, the expression *`eval<zero_constv_q,Ts...>`* will *always* reduce to `const_q`: the fold's combining operation curries successive arguments until an invalid set is formed; or until there are no more.

To define a metafunction which is valid *only* for nullary invocations, class template specialisation is necessary. Line 1 in the code below *declares* a primary variadic class template, `zero_const`; which is then specialised on line 2 to return `const_q` for zero arguments. In this scenario, the operational semantics of `eval_v`'s underlying fold will evaluate `zero_const_q` rather than curry further arguments; ensuring `const_q` is now the active metafunction. Hence, `eval<zero_const_q,int,bool>` will reduce to `int`.

```
1 template <class...> struct zero_const;
2 template <        > struct zero_const<> : id<const_q> {};
3
4 using zero_const_q = quote_c<zero_const>;
```

4.3 Explicit Fixed Arity

Our third implementation of the expression evaluator, `eval_n`, allows the user to curate a selection of variadic metafunctions, or metafunction classes; but requires that each is annotated with a chosen arity.

The class template `bases`, defined below, allows an existing variadic metafunction class *`F`*, to be paired with an arity type *`N`*, as a fellow base class, as in *`bases<F,N>`*. For example, `constv_q` (Appendix A.5) could be given an arity of 7 using `bases<constv_q,ic<7>>`.

```
template <class... Ts> struct bases   : Ts... {};
template <>            struct bases<>         {};
```

Expression evaluation is then undertaken through the `eval_n` combinator shown below. The binary metafunction classes `curry_invoke_peekn_q` and `invoke_ifn_t` are defined in Appendices A.2 and A.3. Intuitively, the operation of `curry_invoke_peekn_q` is much like `curry_invoke_peek_q`. Now, however, the decision to apply `invoke` rather than `curry` is determined simply: the arity of a metafunction's representation is decremented with each curried argument. If that arity becomes zero, use `invoke`.

```
template <class... Fs>
using eval_n =
  invoke_ifn_t<invoke<ifoldl_q,curry_invoke_peekn_q,id_q,Fs...>>;
```

A disadvantage of this approach is that the user is burdened with the task of manually adding rank values. Certainly a future iteration of Curtains could infer the rank attribute with fixed arity metafunctions. Nevertheless, it is likely that more extensive benefits of this approach will stay muted while template metaprograms remain *untyped*. C++ Concepts [3] should shed light here, and further work will seek to explore Curtains' relationship with dependent types [23]; allowing for example, fixed length vectors, and the generic *zipWith* family of combinators.

5 Using the Curtains API

The method of evaluating a type expression involving metafunction classes and other types was introduced via the `eval` combinator in Sections 3 and 4. We now consider examples from the perspective of the end user of the *Curtains* API.

5.1 Defining Metafunctions using Equations

Notably, the application of a metafunction class via `eval` will be curried *implicitly*. For example, `eval<const_q,char>` produces a curried, unevaluated metafunction class, which can then be applied to a second type argument; producing the final result: `eval<eval<const_q,char>,bool>` ≡ `char`. We can then comfortably define a metafunction for the composition of two metafunctions; as shown below.

```
template <class F, class G>
struct compose_t
{
  template <class T>
  using m_invoke = eval<F,eval<G,T>>;
};
using compose_q = quote<compose_t>;
```

Thanks to the implicit currying of the `eval` combinator, the composition will *also* work when given non-unary metafunctions `F` and `G`. For example, a comparable C++ template metaprogram expression to Haskell's *((.) const id 1 2)* is `eval<compose_q,const_q,id_q,int,char>`; which reduce to *1* and `int` respectively.

We now highlight a new factor in the definition of higher-order metafunctions, concerning a flexibility in the syntax. The definition of `compose_t` shown above is a *binary* metafunction which returns a *unary* metafunction via a nested alias template named `m_invoke`. This is analogous to the Haskell definition of the composition operator *(.)* shown below:

```
(.) f g = \x -> f (g x)
```

As with the C++ metafunction `compose_t`, syntactically this defines a binary function which returns a unary lambda function. Note that it is also sometimes convenient to define such functions using a shorter, "equational syntax":

```
(.) f g x = f (g x)
```

This flexibility is found in most languages with lambda expressions; though C++ metaprogramming does not support the latter form. Curtains' support for implicit currying *does* however permit such equational definitions. The alternative definition for `compose_t` is shown below:

```
template <class F, class G, class T>
using compose_t = eval<F,eval<G,T>>;

using compose_q = quote<compose_t>;
```

This is especially convenient for C++ metaprogramming. Firstly, the "lambda syntax" for C++ higher-order metafunctions is verbose. Secondly, while Curtains uses the name `m_invoke`, there is no standard naming convention for this; the returned metafunction.

5.2 Structured Recursion

This integration of currying facilitates a direct transfer of functional programming idioms to C++ template metaprogramming; especially when manipulating higher-order metafunctions. The derivation of functionality from structure can be demonstrated by the use of recursion schemes including catamorphisms and anamorphisms to create metaprogram equivalents of many familiar functions. A Curtains right-fold definition is shown below, along with a simple type list[14].

```
template <class...> struct list {};

template <class, class, class> struct foldr;
using foldr_q = quote_c<foldr>;

template <class F, class Z, class T, class... Ts>
struct foldr<F,Z,list<T,Ts...>>
     : id<eval<F,T,eval<foldr_q,F,Z,list<Ts...>>>> {};

template <class F, class Z>
struct foldr<F,Z,list<>> : id<Z>                    {};
```

The `foldr` metafunction is constructed from two class template specialisations; corresponding to the traditional pair of defining equations. A Haskell expression such as *(foldr id 42 [id])* reduces to *(id id 42)* $\equiv$ *(42)*. Such an operation is accomplished through the elementary treatment of all functions as unary, with left-associative application; possibly returning another function through currying. Given *id*, the type of the second argument, $(a \to b \to b)$, resolves with a as

[14] A `std::tuple` would do, though the minimal `list` class template shown is sufficient for compile-time calculations.

a function type ($c \to c$). Curtains makes no interpretation of the types/kinds of a metafunction's arguments, but here usefully places no demand on the fold's binary combining operation to be provided with two arguments. Ultimately, an isomorphic Curtains expression, such as `eval<foldr_q,id_q,char,list<id_q>>`, reduces similarly to `char`.

With the simple list-forming metafunction `cons_q` provided in Appendix A.4, metafunctions constructed from `foldr_q` can be defined; with varying levels of effort. For example `eval<foldr_q,cons_q,list<>>` behaves as the identity metafunction when provided with a list argument. A fold can also produce the familiar *map* function in Haskell:

map f = foldr ((:) . f) []

The only difference required for a Curtains definition of *map* is for the infix composition operator to be applied prefix:

```
template <class F>
using map_t = eval<foldr_q,eval<compose_q,cons_q,F>,list<>>;
using map_q = quote<map_t>;
```

A Haskell function to reverse a list using a right-fold is shown below:

reverse xs = foldr (λx y → y . ((:) x)) id xs []

Preparing an equivalent list reversal in Curtains, and mindful of the lack of lambda metafunctions, we may consider class or alias templates. A *point-free* equivalent of the lambda function can instead be created, *(flip (.) . (:))*[15], and is found on line 4 of the complete Curtains *reverse* implementation below[16]:

```
1 template <class L>
2 using reverse_t = eval<
3                     foldr_q,
4                     eval<compose_q,eval<flip_q,compose_q>,cons_q>,
5                     id_q,
6                     L,
7                     list<>
8                   >;
9 using reverse_q = quote<reverse_t>;
```

Tools such as the Pointfree.io website can in fact produce an entirely point-free Haskell list reversal; see `reverse_pf_q` in Appendix A.6 for the code listing. In fact, preparing arbitrarily complicated fold operations by this approach becomes somewhat mechanical. Appendix A.9 includes a Curtains implementation of the Ackermann function, constructed using `foldr_q`.

5.3 The Strict Fixed-point Combinator

A lazy language such as Haskell allows a concise definition of the fixed-point combinator:

[15] The Pointfree.io website is an excellent resource for producing such translations.
[16] A Curtains definition of the Haskell *flip* combinator is provided in Appendix A.4.

fix f = f (fix f)

However, as in traditional C++ template metaprogramming, expression evaluation in Curtains is *eager*. In eager functional languages, an η-expanded definition of the fixed-point combinator can be constructed, wherein the evaluation of *(fix f)* on the right-hand side is delayed when only a single argument is provided to the combinator. This strict form of the fixed-point combinator, sometimes referred to as the Z combinator, can be defined, say in OCaml, as follows:

let rec fix f x = f (fix f) x;;

Curtains adopts exactly the same approach. As usual, an alias template name cannot appear on the right-hand side of its definition; and only a class template can have a forward declaration; which explains the formulation shown below:

```
template <class,class> struct fix_c;

using fix = quote_c<fix_c>;

template <class F, class X>
struct fix_c : id<eval<F,eval<fix,F>,X>> {};
```

Curry's Y combinator defines a fixed point combinator without recursion. This too can be constructed as an η-expanded version of its symmetric form where $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, becomes $\lambda f.\lambda g.(\lambda x.\lambda a.f(xx)a)(\lambda x.\lambda a.f(xx)a)g$. The Curtains version is shown below. The non-recursive factorial function from Appendix A.10 means `eval<fix,fix_fact,ic<3>>` ≡ `eval<y,fix_fact,ic<3>>` ≡ `ic<6>`.

```
template <class F, class X, class A>
using y_helper_t = eval<F,eval<X,X>,A>;
using y_helper   = quote<y_helper_t>;

template <class F, class G>
using y_t = eval<eval<y_helper,F>,eval<y_helper,F>,G>;
using y   = quote<y_t>;
```

6 Related Work

The use of C++ templates and macros for metaprogramming started with Unruh's code that emits some prime numbers as warning messages [36]. Veldhuizen introduced expression templates to the world of C++ metaprogramming [39]. Austern [6] exemplified some commonalities between the STL (the generic programming part of the C++98 [1] and C++11 [2] standard libraries) and functional programming. Alexandrescu [5] presented a *tour de force* of C++ metaprogramming and was the first to identify similarities between that and functional programming. Abrahams and Gurtovoy devoted their book [4] to the metaprogramming libraries of the Boost C++ library.

Golodotz [13] offers a tour on the functional programming nature of C++ metaprogramming by showing how to implement certain metaprograms by mimicking the respective Haskell programs. Sipos et al. [35] informally describe a method for systematically producing metafunctions out of functions written in the pure functional programming language Clean [37]. They advertise that their `Eval` metafunction evaluates the produced metaprograms according to the operational semantics of Clean. As detailed in [15], whilst they do not formally present their operational semantics, their informal explanation suggests remarkable differences between the operational semantics of Clean and that of theirs.

Sinkovics [30] offers certain solutions for improving the functional programming support in Boost.MPL and discusses why they are needed. Sinkovics and Porkoláb [32] advertise implementation of a λ-library on top of the operational semantics of Sipos et al. for embedded functional programming in C++. They also later advertise [28] extension of their λ-library to full support for Haskell. Sinkovics [31,34] offers a restricted solution for emulating let-bindings and explicit currying in template metaprogramming.

Haeri and Schupp [15] demonstrate a real-world exemplification of C++ metaprogramming being functional in nature; exploring impediments against fully automatic cross-lingual development between C++ metaprogramming and Haskell. Armed with that, they suggest further examination of semi-automatic cross-lingual development between C++ metaprogramming and hybrid functional programming languages. Haeri et al. [16] examine that suggestion for Scala and F♯. Lincke et al. [19] discuss a real-world semi-automatic translation from Haskell specifications into efficient C++ metaprograms.

Milewski [26] has a number of posts on his personal blog that speak about Monads, their benefits for C++, and how to implement Monadic entities in C++. He also explains how Monads in Haskell can help the understanding of Boost.Proto – one of the most complicated C++ metaprogramming libraries. Moreover, he has a post on how template metaprogramming with variadic templates is similar to lazy list processing in Haskell. Finally, Sankel [29] shows how to implement algebraic datatypes in C++.

Developments in C++ since 2011 have revolutionised metaprogramming. Variadic templates; generalised constant expressions (`constexpr`); alias templates; and `constexpr-if` have made an especially notable impression. Louis Dionne's influential Hana library [9], now included with the C++ Boost libraries, exploits `constexpr`, with richly typed values allowing both runtime and compile-time overloading through a highly distinctive though traditional syntax. Eric Niebler exploits C++11 features in his 3500 line Meta library [27], which utilises variadic templates and demonstrates some support for explicit currying. Numerous other metaprogramming libraries focus on distinctive aspects, including performance [8]; or evaluation schemes, with Metal [10] originally using implicitly lazy evaluation; though now using eager evaluation. With subsets of these libraries now submitted regularly to Boost efforts have also been made [11] to include common patterns within the standard C++ runtime library.

7 Conclusion

C++ template metaprogramming is an expressive, Turing-complete language which holds the potential to engineer libraries and embedded domain-specific languages supported by compile-time formal verification. In this paper we have introduced the Curtains API which provides a model of higher-order functional programming with implicit currying for C++ template metaprograms, and which aligns with norms adopted by languages such as Haskell, OCaml and F♯.

Three distinct schemes are implemented and described in Sections 3 and 4. The first, and simplest, supports only fixed arity metafunctions; the second supports variadic and nullary metafunctions; while the third can use variadic metafunctions, though only when each instance is annotated with an arity.

With the hope of wider uptake, and of further research, our implementation has utilised structured recursion, permitting a concise, 30 line implementation for the simplest of the three schemes described; and 50 lines for the most complex. The choice of the fold's binary combining operation alone accounts for the difference in implementation between each of the three approaches.

A practical introduction to the API is included in Section 5, highlighting the library's accommodation of a distinct equational definition syntax; as well as demonstration of the potential for implicit currying to enable the use of structured recursion operators, such as *map* and *fold*; as opposed to the explicit recursion more commonly employed.

In future we intend to prioritise C++ Concepts [3] integration. Concepts allow a form of type checking for templates, and are implemented as an extension within GCC since version 6.1. We believe Concepts can assist users of Curtains with numerous concerns, including the typing of metafunction classes and their parameters; and a consequential improvement to error messages. We expect this should also support our aim to include support for Haskell-style type classes. Future work will also introduce support for infix alphanumeric operators with specified associativity and precedence.

A Appendix A

A.1 Quotation for a Class Template Argument

```
template <template <class...> class M>
struct quote_c {
  template <class... Ts>
  using m_invoke = typename iv1430<void,M,Ts...>::type::type;
};
```

A.2 Curry-invoke with Arity

```
template <class, class>
  struct curry_invoke_peekn;

template <class F, class T, auto N>
struct curry_invoke_peekn<bases<F,ic<N>>,T> :
id<bases<curry<F,T>,ic<N-1>>> {};

template <class F, class T>
struct curry_invoke_peekn<bases<F,ic<0>>,T>
     : curry_invoke_peekn<invoke<bases<F,ic<0>>>,T> {};

using curry_invoke_peekn_q = quote_c<curry_invoke_peekn>;
```

A.3 Conditional Invoke with Arity

```
template <class F>
struct invoke_ifn                : id<F>                                {};

template <class F>
struct invoke_ifn<bases<F,ic<0>>> : invoke_ifn<invoke<bases<F,ic<0>>>> {};

template <class F>
using invoke_ifn_t = typename invoke_ifn<F>::type;

using invoke_ifn_q = quote<invoke_ifn_t>;
```

A.4 Additional Utility Metafunctions

```
template <class, class> struct cons;

template <class T, class... Ts>
struct cons<T,list<Ts...>> : id<list<T,Ts...>> {};

using cons_q = quote_c<cons>;

template <class F, class T, class U>
using flip_t = eval<F,U,T>;

using flip_q = quote<flip_t>;
```

A.5 A Sample Variadic Metafunction: `constv_q`

```
template <class T, class...> using constv_t = T;
                             using constv_q = quote<constv_t>;
```

A.6 Point-free Reverse From a Right-Fold

```
using reverse_pf_q = eval<
                       flip_q,
                       eval<
                         foldr_q,
                         eval<compose_q,eval<flip_q,compose_q>,cons_q>,
                         id_q
                       >,
                       list<>
                     >;
```

A.7 Point-free Left-Fold From a Right-Fold

Implementation derived from Hutton [18]; with assistance from the Pointfree.io website; which provides: *(flip . flip foldr id . (flip (.) .) . flip)*.

```
using foldl_q = eval<
                  compose_q,
                  flip_q,
                  eval<
                    compose_q,
                    eval<flip_q,foldr_q,id_q>,
                    eval<
                      compose_q,
                      eval<compose_q,eval<flip_q,compose_q>>,
                      flip_q
                    >
                  >
                >;
```

A.8 The SKI Combinators

```
template <class X, class Y, class Z>
using S_t = eval<X,Z,eval<Y,Z>>;

using I = id_q;
using K = const_q;
using S = quote<S_t>;
```

A.9 Point-free Ackermann Function from a Right-Fold

Here `S` and `const_q` are used in lieu of *(<*>)* and *pure*, for the Applicative instance of *((->) r)*. The implementation is derived from Hutton [18]; with assistance from the Pointfree.io website.

```
using ack = eval<
              foldr_q,
              eval<
                const_q,
                eval<
                  S,
                  eval<S,eval<const_q,foldr_q>,const_q>,
                  eval<flip_q,dollar_q,list<void>>
                >
              >,
              eval<cons_q,void>
            >;
```

A.10 Non-recursive Factorial for use with the Fixpoint Combinator

```
template <class, class>   struct mul_c;
template <auto M, auto N> struct mul_c<ic<M>, ic<N>> : id<ic<M*N>> {};
using mul = quote_c<mul_c>;

template <class F, class N>
struct fix_fact_c          : id<eval<mul,N,eval<F,eval<pred,N>>>> {};

template <class F>
struct fix_fact_c<F,ic<0>> : id<ic<1>> {};

using fix_fact = quote_c<fix_fact_c>;
```

A.11 Primitive Left-Fold from a C++17 Fold Expression

```
template <class T, class>
struct const_ { using type = T; };

template <class T, class U, class F>
auto operator+(const_<T,F>, const_<U,F>) {
  return const_<invoke<F,T,U>,F>{};
}

template <class F, class Z, class... Ts>
struct ifoldl {
  using type =
    typename decltype((const_<Z,F>{} + ... + const_<Ts,F>{}))::type;
};

using ifoldl_q = quote_c<ifoldl>;
```

A.12 Reduction Steps of a Sample Curtains Expression

```
eval<const_q,id_q,int,char>
   ≡ {eval alias template}
invoke<ifoldl_q,curry_invoke_q,id_q,const_q,id_q,int,char>
   ≡ {invoke alias template}
ifoldl_q::m_invoke<curry_invoke_q,id_q,const_q,id_q,int,char>
   ≡ {ifoldl_q alias template}
quote_c<ifoldl>::m_invoke<curry_invoke_q,id_q,const_q,id_q,int,char>
   ≡ {m_invoke alias template member of quote_c template}
iv1430<void,ifoldl,curry_invoke_q,id_q,const_q,id_q,int,char>::type::type
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,id_q,const_q,id_q,int,char>::type
   ≡ {type member of ifoldl specialisation}
ifoldl<curry_invoke_q,invoke<curry_invoke_q,id_q,const_q>,id_q,int,char>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,curry_invoke_q::m_invoke<id_q,const_q>,id_q,int,char>::type
   ≡ {curry_invoke_q alias template}
ifoldl<curry_invoke_q,quote_c<curry_invoke>::m_invoke<id_q,const_q>,id_q,int,char>::type
   ≡ {m_invoke alias template member of quote_c template}
ifoldl<curry_invoke_q,iv1430<void,curry_invoke,id_q,const_q>::type::type,id_q,int,char>::type
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,curry_invoke<id_q,const_q>::type,id_q,int,char>::type
   ≡ {type member of curry_invoke template specialisation}
ifoldl<curry_invoke_q,invoke<id_q,const_q>,id_q,int,char>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,id_q::m_invoke<const_q>,id_q,int,char>::type
   ≡ {id_q alias template}
ifoldl<curry_invoke_q,quote_c<id>::m_invoke<const_q>,id_q,int,char>::type
   ≡ {m_invoke alias template member of quote_c template}
ifoldl<curry_invoke_q,iv1430<void,id,const_q>::type::type,id_q,int,char>::type
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,id<const_q>::type,id_q,int,char>::type
   ≡ {type member of id template}
ifoldl<curry_invoke_q,const_q,id_q,int,char>::type
   ≡ {type member of ifoldl specialisation}
ifoldl<curry_invoke_q,invoke<curry_invoke_q,const_q,id_q>,int,char>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,curry_invoke_q::m_invoke<const_q,id_q>,int,char>::type
   ≡ {curry_invoke_q alias template}
ifoldl<curry_invoke_q,quote_c<curry_invoke>::m_invoke<const_q,id_q>,int,char>::type
   ≡ {m_invoke alias template member of quote_c template}
ifoldl<curry_invoke_q,iv1430<void,curry_invoke,const_q,id_q>::type::type,int,char>::type
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,curry_invoke<const_q,id_q>::type,int,char>::type
   ≡ {type member of curry_invoke primary template}
ifoldl<curry_invoke_q,curry<const_q,id_q>,int,char>::type
   ≡ {type member of ifoldl specialisation}
ifoldl<curry_invoke_q,invoke<curry_invoke_q,curry<const_q,id_q>,int>,char>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,curry_invoke_q::m_invoke<curry<const_q,id_q>,int>,char>::type
   ≡ {curry_invoke_q alias template}
ifoldl<curry_invoke_q,quote_c<curry_invoke>::m_invoke<curry<const_q,id_q>,int>,char>::type
   ≡ {m_invoke alias template member of quote_c template}
ifoldl<curry_invoke_q,iv1430<void,curry_invoke,curry<const_q,id_q>,int>::type::type,char>::type
```

```
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,curry_invoke<curry<const_q,id_q>,int>::type,char>::type
   ≡ {type member of curry_invoke template specialisation}
ifoldl<curry_invoke_q,invoke<curry<const_q,id_q>,int>,char>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,curry<const_q,id_q>::m_invoke<int>,char>::type
   ≡ {m_invoke alias template member of curry template}
ifoldl<curry_invoke_q,invoke<const_q,id_q,int>,char>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,const_q::m_invoke<id_q,int>,char>::type
   ≡ {const_q alias template}
ifoldl<curry_invoke_q,quote<const_t>::m_invoke<id_q,int>,char>::type
   ≡ {m_invoke alias template member of quote template}
ifoldl<curry_invoke_q,iv1430<void,const_t,id_q,int>::type,char>::type
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,const_t<id_q,int>,char>::type
   ≡ {const_t alias template}
ifoldl<curry_invoke_q,id_q,char>::type
   ≡ {type member of ifoldl specialisation}
ifoldl<curry_invoke_q,invoke<curry_invoke_q,id_q,char>>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,curry_invoke_q::m_invoke<id_q,char>>::type
   ≡ {curry_invoke_q alias template}
ifoldl<curry_invoke_q,quote_c<curry_invoke>::m_invoke<id_q,char>>::type
   ≡ {m_invoke alias template member of quote_c template}
ifoldl<curry_invoke_q,iv1430<void,curry_invoke,id_q,char>::type::type>::type
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,curry_invoke<id_q,char>::type>::type
   ≡ {type member of curry_invoke specialisation}
ifoldl<curry_invoke_q,invoke<id_q,char>>::type
   ≡ {invoke alias template}
ifoldl<curry_invoke_q,id_q::m_invoke<char>>::type
   ≡ {id_q alias template}
ifoldl<curry_invoke_q,quote_c<id>::m_invoke<char>>::type
   ≡ {m_invoke alias template member of quote_c template}
ifoldl<curry_invoke_q,iv1430<void,id,char>::type::type>::type
   ≡ {type member of iv1430 template specialisation}
ifoldl<curry_invoke_q,id<char>::type>::type
   ≡ {type member of id template}
ifoldl<curry_invoke_q,char>::type
   ≡ {type member of ifoldl primary template}
char
```

References

1. International Standard ISO/IEC 14882:1998(E): Programming Languages — C++ (1998)
2. International Standard ISO/IEC 14882:2011: Information technology – Programming languages – C++ (2011)
3. International Standard ISO/IEC 19217:2015: Information technology – Programming languages – C++ Extensions for concepts (2015)
4. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. AW Prof. (2004)
5. Alexandrescu, A.: Modern C++ Design: Generic Prog. and Design Patterns Applied. Addison-Wesley Longman Publishing, Boston, MA, USA (2001)
6. Austern, M.H.: Generic Prog. and the STL: Using and Extending the C++ Standard Template Library. AW Prof. Comp. Series, AW Longman Publ. Co. (1998)
7. Brown, W.E.: Proposing Standard Library Support for the C++ Detection Idiom. Tech. rep., ISO WG21 C++ Working Group (April 2015), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4436.pdf
8. Deppe, N., Douwes, C., Fresk, E., Holmes, O., Poelen, J.: Kvasir::mpl (2017), https://github.com/kvasir-io/mpl
9. Dionne, L.: Hana (2013), https://github.com/boostorg/hana
10. Dutra, B.: Metal (2018), https://github.com/brunocodutra/metal
11. Escribá, V.J.B.: P0343R1: Meta-Programming High-Order Functions. Tech. rep., ISO WG21 C++ Library Evolution Working Group (2016)
12. Gil, J., Gutterman, Z.: Compile Time Symbolic Derivation with C++ Templates. In: Proceedings of the 4th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4. pp. 18–18. COOTS'98, USENIX Association, Berkeley, CA, USA (1998), http://dl.acm.org/citation.cfm?id=1268009.1268027
13. Golodotz, S.: Functional Programming Using C++ Templates (Part 1) (October 2007), http://accu.org/index.php/journals/1422
14. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010), http://eigen.tuxfamily.org
15. Haeri, S.H., Schupp, S.: Functional Metaprogramming in C++ and Cross-Lingual Development with Haskell. Tech. rep., Uni. Kansas (Oct 2011), draft Proc. 23^{rd} Symp. Impl. and Appl. Func. Langs., ITTC-FY2012-TR-29952012-01
16. Haeri, S.H., Schupp, S., Hüser, J.: Using Functional Languages to Facilitate C++ Metaprogramming. In: Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming. pp. 33–44. WGP '12, ACM (2012)
17. Holmes, O., Kurdej, M., Poelen, J.: Brigand Meta-Programming Library (2015), https://github.com/edouarda/brigand
18. Hutton, G.: A Tutorial on the Universality and Expressiveness of Fold. Journal of Functional Programming **9**(4), 355–372 (Jul 1999)
19. Lincke, D., Schupp, S., Ionescu, C.: Functional Prototypes for Generic C++ Libraries: A Transformational Approach Based on Higher-Order, Typed Signatures. Int. J. Soft. Tools for Tech. Transfer **17**(1), 91–105 (Feb 2015), https://doi.org/10.1007/s10009-014-0299-0
20. Lumsdaine, A., Siek, J., Lee, L.Q.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
21. Mach, S.: Metatrace (2010), https://github.com/phresnel/metatrace
22. Marlow, S.: Haskell 2010 Language Report (2010)

23. McBride, C.: Faking It: Simulating Dependent Types in Haskell. Journal of Functional Programming **12**(5), 375–392 (Jul 2002)
24. Meijer, E., Fokkinga, M., Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture. pp. 124–144. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
25. Merrill, J.: C++ Core Issue 1430 (2011), http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1430
26. Milewski, B.: Bartosz Milewski's Programming Cafe, http://bartoszmilewski.wordpress.com
27. Niebler, E.: Meta: A Tiny Metaprogramming Library (2014), https://ericniebler.github.io/meta/index.html
28. Porkoláb, Z., Sinkovics, Á.: C++ Template Metaprogramming with Embedded Haskell. In: Proc. 8th Int. Conf. Generative Prog. & Component Engineering (GPCE 2009). pp. 99–108. ACM, New York, NY, USA (2009)
29. Sankel, D.: Algebraic Data Types Series, C++Next: The next generation of C++, http://cpp-next.com/archive/2010/07/algebraic-data-types/
30. Sinkovics, Á.: Funcional Extensions to the Boost Metaprogram Library. In: Proc. 2^{nd} Works. Generative Tech. Elec. Notes in Theo. Comp. Sci., vol. 264, pp. 85–101 (Jul 2011)
31. Sinkovics, Á.: Nested Lamda Expressions with Let Expressions in C++ Template Metaprograms. pp. 63–76 (2011)
32. Sinkovics, Á., Porkoláb, Z.: Expressing C++ Template Metaprograms as Lambda Expressions. In: TFP. pp. 1–15 (2009)
33. Sinkovics, Á., Porkoláb, Z.: Metaparse: Compile-time Parsing with Template Metaprogramming. Aspen, USA (2012), https://github.com/boostcon/cppnow_presentations_2012/blob/master/papers/metaparse_paper.pdf
34. Sinkovics, A.: Functional Extensions to the Boost Metaprogram Library. Electronic Notes in Theoretical Computer Science **264**(5), 85–101 (Jul 2011), https://doi.org/10.1016/j.entcs.2011.06.006
35. Sipos, Á., Porkoláb, Z., Pataki, N., Zsók, V.: Meta<Fun>: Towards a Functional-Style Interface for C++ Template Metaprograms. Tech. rep., Eötvös Loránd Uni, Fac. of Inf., Dept. Prog. Langs., Pázmány Péter sétány 1/C H-1117 Budapest, Hungary (2007)
36. Unruh, E.: Prime Number Computation (1994), ANSI X3J16-94-0075/ISO WG21-462
37. van Eekelen, M., M., d.: Mixed Lazy/Strict Graph Semantics. In: Grelck, C., Huch, F. (eds.) Impl. and Appl. of Func. Langs., 16th Int. W., IFL04. pp. 245–260. Tech. Rep. 0408, Christian-Albrechts-Universität zu Kiel, Lüebeck, Germany (Sep 2004)
38. Vandevoorde, D., Josuttis, N.M., Gregor, D.: C++ Templates: The Complete Guide, Second Edition. AW Prof. (2018)
39. Veldhuizen, T.L.: Expression Templates. C++ Report **7**(5), 26–31 (1995)
40. Veldhuizen, T.L.: Scientific Computing: C++ Versus Fortran: C++ Has More Than Caught up. Dr. Dobb's Journal of Software Tools **22**(11), 34, 36–38, 91 (Nov 1997)